An Improved Parallel Singular Value Algorithm and Its Implementation for Multicore Hardware

Azzam Haidar
Electrical Engineering and Computer Science
University of Tennessee
Knoxville, Tennessee, USA
haidar@eecs.utk.edu

Jakub Kurzak
Electrical Engineering and Computer Science
University of Tennessee
Knoxville, Tennessee, USA
kurzak@eecs.utk.edu

Piotr Luszczek
Electrical Engineering and Computer Science
University of Tennessee
Knoxville, Tennessee, USA
luszczek@eecs.utk.edu

ABSTRACT

The enormous gap between the high-performance capabilities of today's CPUs and off-chip communication poses extreme challenges to the development of numerical software that is scalable and achieves high performance.

In this article, we describe a successful methodology to address these challenges—starting with our algorithm design, through kernel optimization and tuning, and finishing with our programming model. All these lead to development of a scalable high-performance Singular Value Decomposition (SVD) solver. We developed a set of highly optimized kernels and combined them with advanced optimization techniques that feature fine-grain and cache-contained kernels, a task based approach, and hybrid execution and scheduling runtime, all of which significantly increase the performance of our SVD solver.

Our results demonstrate a many-fold performance increase compared to currently available software. In particular, our software is two times faster than Intel's Math Kernel Library (MKL), a highly optimized implementation from the hardware vendor, when all the singular vectors are requested; it achieves a 5-fold speed-up when only 20% of the vectors are computed; and it is up to 10 times faster if only the singular values are required.

Categories and Subject Descriptors

G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Singular value decomposition, Eigenvalues and eigenvectors*

Keywords

Singular Value Decomposition, eigenvalues and eigenvectors, Performance, Reduction to bidiagonal, task parallelism

1. INTRODUCTION

Standard Singular Value Decomposition (SVD) problem [27] for a given matrix $A \in \mathbb{R}^{m\times n}$ (or $A \in \mathbb{C}^{m\times n}$) finds a diagonal matrix $\Sigma \in \mathbb{R}^{m\times n}$ (or $\Sigma \in \mathbb{C}^{m\times n}$) and orthogonal (or unitary) matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ (or $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{n\times n}$) such that $A = U\Sigma V^\top$ (or $A = U\Sigma V^H$). Matrices U and V are unique up to the signs of respective columns when $m = n$ and the singular values are unique [58]. Also, the decomposition is always possible [26, 28]. The diagonal elements of Σ are singular values of A, the columns of U are called its left singular vectors, and the columns of V are called its right singular vectors.

The necessity of calculating SVDs emerges from various computational science and engineering areas, e.g., in statistics where it is directly related to the principal component analysis method [40, 41], in signal processing and pattern recognition as an essential filtering tool, and in analysis of control systems [57]. Also, the SVD plays a very important role in linear algebra. It has applications in such areas as least squares problems [26, 28, 51], in computing the pseudoinverse [26], in computing the Jordan canonical form [29]. In addition, SVD is used in solving integral equations [39], in digital image processing [4], in information retrieval [45], in seismic reflection tomography [10, 23], and in optimization [5].

SVD decomposition of a dense matrix is computed by following the decompositional approach to matrix computation [59]. First, the dense matrix A is transformed to an upper bidiagonal form B by applying successive distinct orthogonal transformations [42] from the left (Q) as well as from the right (P): $B = Q^\top AP$ (or $B = Q^H AP$). This reduction step is called *bidiagonal reduction* (BRD for short), and it has always been the most time-consuming phase when computing the SVD. The next phase after BRD is the process of obtaining the singular values using the *divide-and-conquer iteration*. Finally, calculation of the corresponding singular vectors from the reduced form is performed using either the dqds algorithm [24] or using Cuppen's divide-and-conquer algorithm [32, 44] of *divide-and-conquer back-transformation*. The BRD portion of the computation can easily consume over 90% of the time needed to obtain the singular values [53], and roughly 70% if singular vectors are additionally calculated [53]. The QR iteration [17, 18] is no longer a method of choice for singular vectors because it takes roughly 50% longer than the methods mentioned earlier. The optimization techniques we present here are only applicable to sqaure matrices because a slightly different approach is necessariy for the rectangular ones [54].

This article is organized as follows: Section 2 gives an overview of the related work in the field; Section 3 summarizes the main contributions of this work; Section 4 provides sufficient background information for the techniques we describe later on; Section 5 presents detail description of the stages of our reduction algorithm and its implementation; Section 5.4 details the computation of singular vectors; Section 6 shows the results from our experiments. Finally, Section 7 concludes the article and suggests potential future directions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13 November 17-21, 2013, Denver, CO, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2378-9/13/11 ...$15.00
http://dx.doi.org/10.1145/2503210.2503292 .

2. RELATED WORK

A two-step reduction for the generalized symmetric eigenvalue problem was reported for the first time in the context of an out-of-core solver [30, 31]. Later, the two-stage approach [7, 49] was generalized to a multi-stage implementation [8] to reduce a matrix to tridiagonal, bidiagonal, and Hessenberg forms. It has provided much needed increase in the performance of the said routines. The actual number of stages necessary to reduce the matrix to the corresponding form is a tunable parameter, which depends on the underlying hardware architecture. The general idea is to cast expensive memory operations, occurring during the panel factorization into fast compute intensive ones. This general framework is called Successive Band Reductions (SBR) [8]. SBR is used to reduce a symmetric dense matrix to tridiagonal form, which is required to solve the symmetric eigenvalue problem (SEVP). This toolbox applies two-sided orthogonal transformations to the matrix based on Householder reflectors and successively reduces the matrix bandwidth size until a suitable one is reached. The off-diagonal elements are then annihilated column-wise, which produces large fill-in blocks or bulges to be chased down, and therefore, may result in substantial extra flops. If eigenvectors are additionally required, the transformations can be efficiently accumulated using Level 3 BLAS operations to generate the orthogonal matrix. It is also worth noting that the SBR package relies heavily on multithreaded optimized BLAS to achieve parallel performance [50]. We call this a *fork-join* paradigm which is a variant of Bulk Synchronous Parallel (BSP) [61].

PIRO_BAND toolbox [16] implemented a similar technique which only focusses on the last stage, i.e., the reduction from band form to the condensed structures. This software enables us to reduce, not only symmetric band matrices to tridiagonal form but also non-symmetric band matrices to bidiagonal form needed for the symmetric eigenvalue problem and the singular value decomposition, respectively. This sequential toolbox employs fine-grained computational kernels, since it only operates on regions located around the diagonal structure of the matrix. However, the off-diagonal entries are annihilated element-wise and the number of fill-in elements is drastically reduced compared to the SBR implementation. As a consequence, the overall time to solution has been improved compared to SBR package, even though the PIRO_BAND implementation is purely sequential. Finally, PIRO_BAND relies on pipelined plane rotations (i.e., Givens rotations) to annihilate the off-diagonal entries.

The two-stage approach was applied to the TRD (Triangular Reduction) [34] and to SVD [35, 53, 54] in combination with tile algorithms and runtime scheduling based on data dependences between tasks that operate on the tiles. This resulted in very good performance but has never been used to compute the singular vectors. Obtaining the eigen vectors through the two-stage technique takes away some of the performance gains when applied to the symmetric eigenvalue problem [37], and in this article we investigate whether this remains true for the SVD.

More recently, a new parallel, high-performance implementation of the tile BRD algorithm on homogeneous multicore architectures was introduced [53]. It used a two-stage approach and it did away with the BSP paradigm in favor of bringing the parallelism to the fore, but hid the problems of concurrency with the use of a runtime scheduler that keeps track of data dependences. The first stage reduces the matrix to band tridiagonal form, and uses high compute intensive kernels. In this article, we improve upon these kernels to remove scheduler and cache-miss overheads. The second stage follows the SBR principle of annihilating the extra entries column-wise. The dynamic runtime system was used to schedule the computational tasks in a fashion, that preserves data dependence, and maximizes the benefits of parallel execution. Here, we further the development and optimization of this work by using a more efficient bulge-chasing stage and adding the computation of singular vectors.

To the best of our knowledge, this technique has not been applied in combination with the Multiple Relatively Robust Representations (MRRR) algorithm, which still continues to be work in progress [62].

An alternative approaches for SVD are based the Jacobi method and have been used successfully in the parallel setting for dense matrices [6, 56].

3. RESEARCH CONTRIBUTIONS

Besides the software development efforts that we investigate to accomplish an efficient implementation, we highlight three main contributions related to the algorithm's design:

• **High performance fine grained memory aware and computationally intense tasks.** Our goal to efficient hardware usage and parallelism relies on splitting the computation into tasks that either increase computational intensity or reduce data movement. Two main issues should be taken into consideration here. First, the task splitting and determination of granularity is essential for obtaining high performance. Moreover, the data reuse among the CPU-cores should also be taken into consideration to minimize communication and achieve good performance. For that, despite the use of new kernels that have been developed by Luszczek et al. [55], for the first stage of the reduction to bidiagonal, we also developed new fine-grained and memory-aware BLAS kernels to be used for the second stage of the BRD reduction (i.e., the bulge chasing procedure), and for the update of the singular vectors by the transformation matrices. Sections 5.2 and 5.4 provide more information about those new fine-grained and cache-friendly numerical kernels.

• **Mapping computation to hardware via hybrid scheduling.** We developed our algorithm in such a way to map computational tasks to the strengths of the available hardware components, taking care of the data reuse. Our algorithm also uses techniques to mix between dynamic and static scheduling to extract efficiency and performance. The impact of this technique is well observed during either the bulge chasing stage or the singular vectors update. First, the bulge chasing stage operates on a small amount of data of the size $b \times n$, where b and n are respectively the band width and the size of the band matrix. Also, most of the operations are memory bound and the parallelism is limited. Hence, it is better sometimes to let this stage run on a small number of cores while increasing data locality rather than to let all the cores work while increasing the data movement. Second, the application of the Householder reflectors generated by the bulge chasing stage are very complicated and rely on sequential overlaps between them. We combine the computation splitting (a technique based on the available number of resources and on the size of the level 2 cache) with hybrid task scheduling. This combination is the defining factor that determines the block size required for data reuse, and the way in which parallelism is extracted for high performance when such bandwidth-bound operations are concerned. Section 5.4 provides further information about these techniques.

• **Examining the trade-off between performance and extra computation.** A proper use of this trade-off reduces overall execution time, which we believe will become increasingly important for the current and the up-coming hardware designs. An advanced optimization strategy, which consists of aggregating the applications of Householder reflectors occurring within a single data block, while adding a small extra cost is employed to remove the communications overhead as well as to enhance the memory reuse for obtaining high performance algorithms.

4. BACKGROUND

Tile algorithms are based on the idea of processing the matrix by square tiles of relatively small size (between 100×100 and 200×200 elements). The rationale is that the one, two, or three tiles, involved in a particular matrix operation fit entirely in some level of the cache hierarchy, and *capacity* cache misses are eliminated. Tiling is the principal optimization technique in optimizing the performance of the basic dense linear algebra operation of matrix multiplication, and comes directly from the fundamental compiler optimization of *loop tiling*. The motivation for deriving the class of tile algorithms came from the desire to extend the same performance benefits to dense matrix factorizations. At the same time, tile algorithms allow to easy expression of the algorithm in the form of a task graph or *Direct Acyclic Graph* (DAG), suitable for dynamic runtime scheduling using dataflow principles [12, 36, 48]. The algorithms are extremely efficient when matched with a corresponding matrix layout [33], as described in the following section.

The benefit of using tile algorithms on multicore processors was initially demonstrated for the Cholesky factorization [46], which did not require any algorithmic modifications. The tile approach was then extended to the LU and QR factorizations [13, 14, 47], by introducing the idea of incremental panel factorization, where the panel is factored by descending down the matrix tile by tile. In the case of the LU factorization, numerical properties are affected, because of the use of elemental transformation. In the case of the QR factorization, numerical properties are not significantly affected, due to the use of Householder reflections, which are orthogonal transformations.

The idea of applying Householder transformations by tiles can easily be extrapolated to the bi-diagonal and tri-diagonal reductions, allowing for the construction of very efficient algorithms for the solution of the singular value problem and the symmetric eigenvalue problem. The caveat is that the reductions can be done easily to a band form, instead of the proper bi-diagonal matrix or a tri-diagonal matrix (with a single subdiagonal). The solution is to reduce to the band form first, and then produce the proper form through the process of *bulge chasing*, i.e., successive elimination of the subdiagonal entries by a series of Householder transformations [34, 35, 52–55]. Because both the reduction to the band form and the bulge chasing process can be implemented in a parallel and cache-efficient manner, the two-stage procedure is an order of magnitude faster than the legacy approach of LAPACK, which relies heavily on Level 2 BLAS operations, is memory bound, and therefore inefficient.

It is always beneficial for performance to couple the algorithm with a data layout that matches the processing pattern. For tile algorithms, the corresponding layout is the *tile layout*, developed by Gustavson et al. [33] and shown in Figure 1. The matrix is arranged in square submatrices, called tiles, where each tile occupies a contiguous region of memory. The particular type of layout used here is referred to as *Column-Column Rectangular Block* (CCRB). In this flavor of the tile layout, tiles follow the column-major order and elements within tiles follow the column-major order. The same applies to the blocks A_{11}, A_{21}, A_{12}, and A_{22}.

Because the entire matrix occupies a contiguous region of memory, translation between the tile layout and the legacy FORTRAN 77 layout can be done in place, without changing the memory footprint. Gustavson et al. [33] devised a collection of routines for performing this translation in a parallel and cache efficient manner. It is important to observe that the layout translation routines have a broader impact in forming the basis for a fast transposition operation. The codes are distributed as part of the PLASMA library.

Form the standpoint of serial execution, tile layout minimizes *conflict* cache misses, because two different memory locations within

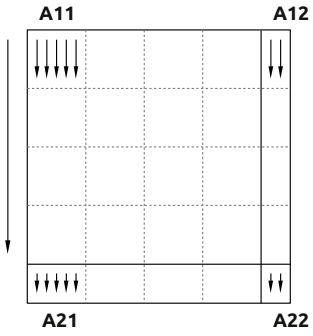


Figure 1: The Column-Column Rectangular Block (CCRB) matrix layout.

the same tile cannot be mapped to the same set of a set-associative cache. The same applies to the *Translation Lookaside Buffer* (TLB) misses. In the context of parallel execution, tile layout minimizes the probability of *false sharing*, which is only possible at the beginning and end of the contiguous memory region occupied by each tile, and can easily be eliminated altogether, if the matrix is aligned with cache lines and tiles are divisible by the cache line size. Tile layout is also beneficial for prefetching, which in the case of strided memory access is likely to generate useless memory traffic.

The tile algorithms dramatically increase the opportunities for scheduling by exposing a much higher level of parallelism and facilitating pipelining of operations. In order to exploit the fine-grained parallelism to its fullest, efficient schedules have to be designed, while data dependencies are preserved, i.e., data hazards are prevented. This has been done for both the simpler single-sided factorizations, such as Cholesky, LU and QR [1, 2, 13, 14, 19–21, 36, 48], as well as the more complicated two-sided factorizations, such as the reductions to band bi-diagonal and band tri-diagonal form [34, 35, 52–55]. The process of constructing such schedules through manipulation of loop indexes and enforcing them by progress tables is tedious and error-prone. Using a runtime dataflow scheduler is a good alternative. Here, a superscalar scheduler is used.

Superscalar schedulers exploit multithreaded parallelism in a similar way as superscalar processors exploit *Instruction Level Parallelism* (ILP). Scheduling proceeds under the constraints of data hazards: *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). In the context of multithreading, superscalar scheduling is a way of automatically parallelizing serial code. The programmer is responsible for encapsulating the work in side-effect-free functions (parallel tasks) and providing directionality of their parameters (input, output, input-and-output), and the scheduling is left to the runtime. Scheduling is done by conceptually exploring the *Directed Acyclic Graph* (DAG), or task graph, of the problem. In practice the DAG is never built entirely, and instead explored in a *sliding window* fashion. The superscalar scheduler used here is the *QUeuing And Runtime for Kernels* (QUARK) [63] system, developed at the University of Tennessee.

5. MULTI-STAGE ASYNCHRONOUS ALGORITHM FOR BRD

Due to its high computational complexity of $O(\frac{8}{3}n^3)$ (for square matrices) and interdependent data access patterns, the bidiagonal reduction phase is the most challenging stage to develop and optimize: both algorithmically and from the implementation standpoint. There are two main approaches targeting the problem: the standard

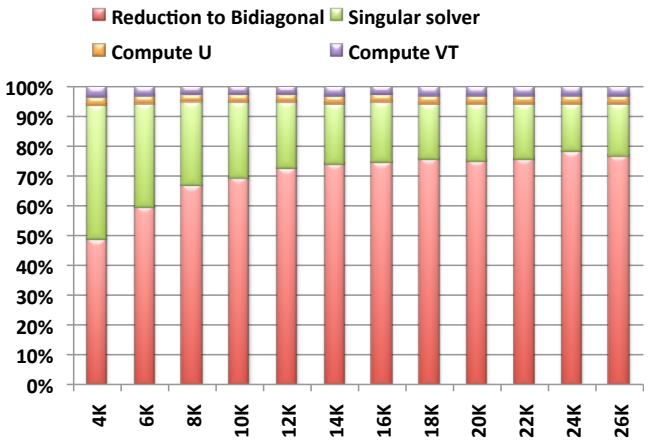
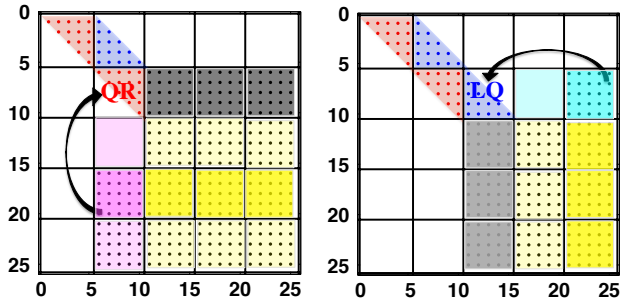


Figure 2: The percentage of the time spent in each kernel of the DGESDD solver using the standard one stage approach to compute the bidiagonal form.



(a) QR factorization of tile $A_{2,2}$ (b) LQ factorization of tile $A_{2,3}$

Figure 3: Kernel execution of the BRD algorithm during the first stage.

one-stage approach from LAPACK [3], whereby the Householder transformations are grouped and applied in a blocked fashion to directly reduce the dense matrix to bidiagonal form. A more recent approach, the two-stage one [34], applies blocked Householder transformations to, first, reduce the matrix to a band form, and, second, uses the bulge chasing technique to reduce the band matrix to the canonical bidiagonal form.

The one-stage reduction to bidiagonal form, as it is implemented in LAPACK, suffers from lack of efficiency. To understand it better, one has to focus on the two computational steps that are repeated until the matrix is reduced: the panel factorization and the update of the trailing submatrix. First, the panel factorization computes the similarity transformations (Householder reflectors) to introduce zeros to the entries below the subdiagonal within a single block of columns. This step is memory-bound because each reflector relies on two matrix-vector multiplications with the trailing submatrix. The step is thus critical as the entire trailing submatrix needs to be loaded into memory and very few floating-point operations are executed on all the transfered data. As the memory bandwidth becomes more limited and does not scale with the number of cores, the panel factorization step will not scale for large matrices that do not fit in cache. Consequently, any implementation is bound to generate a tremendous amount of cache and TLB misses. The trailing submatrix update applies the blocked reflectors using a compute-bound operations that utilize Level 3 BLAS, which have high data reuse ratio allowing for highly tuned implementations. In addition, Level 3 BLAS lend themselves to parallelization due to their inherent data locality properties that can be exploited to achieve high performance rates for large amounts of operations with most of it completely independent and thus perfectly suited for multicore processors. Simply put, this is the computational step of the reduction rich in parallelism and high-performance computations. Unfortunately, each panel factorization must be synchronized with the corresponding update of the trailing submatrix, which prevents asynchronous execution and overlap of memory-bound and compute-bound steps that could potentially alleviate the effects of the former. Figure 2, shows the percentage of the total time for each of the four components of the SVD solver using the standard one-stage reduction approach when all the singular vectors are computed. The figure makes it clear that the reduction to the bidiagonal form requires more than about 70% of the total time for the case when all the singular vectors are computed. In addition, when only singular values are needed, the reduction requires about 90% of the total computing time. This was the main motivation for our work, which is to analyze and develop a new algorithm that computes the singular value decomposition that is based on the two-stage approach. We focus on modern multicore architectures and use the technologies available through the PLASMA project [60].

The two-stage reduction is designed to overcome the limitations of the one-stage approach that relies heavily on memory-bound operations. It also increases the use of compute-intensive operations that benefit from the increase in CPU core count. Many algorithms have been extensively studied in the context of the symmetric eigenvalue problem [8, 9] and, more recently, tile algorithms have achieved good performance [34, 55]. The idea behind them is to split the original one-stage approach into a compute-intensive phase (first stage) and a memory-bound phase (second or *bulge-chasing* stage). The first stage reduces the original general dense matrix to a band form (either upper or lower), and the second stage reduces the band form to the canonical bidiagonal form (again, either upper or lower). The two-stage approach that is used in our implementation and exhibits some similarities to what has been developed for the symmetric eigenvalue problem [34]. To put our work in a proper perspective, we start by briefly describing the first stage (reduction from full dense to band), and then we explain in more detail the reduction from band to the bidiagonal form. We will also talk about the scheduling techniques that our implementation relies on.

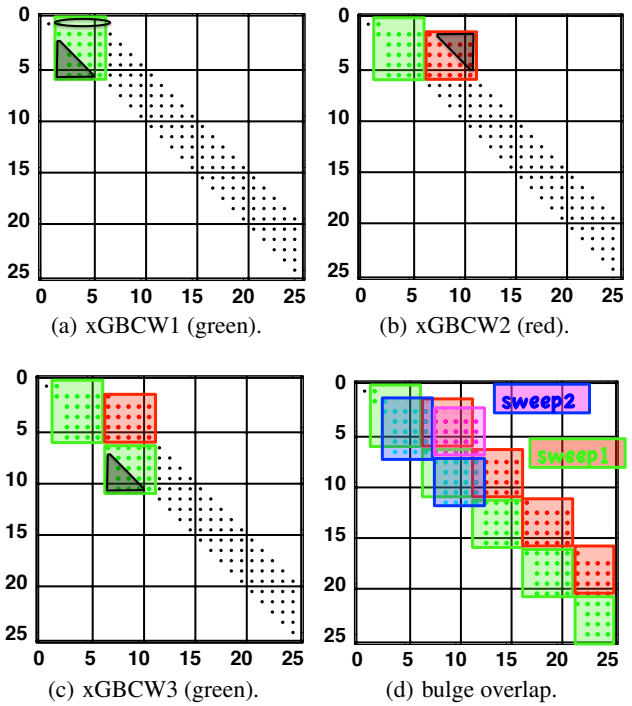
5.1 First Stage: Compute-Intensive and Efficient Kernels

The first stage applies a sequence of blocked Householder transformations to reduce the general dense matrix to either an upper or a lower band matrix. This stage uses compute-intensive matrix-multiply kernels, that eliminate the memory-bound matrix-vector product from the one-stage panel factorization. It also illustrates a beneficial data access patterns through the use of compute-intensive operations based on Level 3 BLAS for large portions of the code [8, 22, 25, 38]. Additionally, this stage is made highly parallel because of the tile algorithm formulation [2] that brings the parallelism to the fore. Conceptually and physically, the matrix is split into $nt \times nt$ tiles ($nt = n/nb$, nb is the size of the tile and the resulting matrix bandwidth). The data within a tile is stored contiguously in memory. The algorithm then proceeds as a collection of interdependent tasks that operate on the tile data layout. Figure 3 highlights the execution breakdown during the second step of the first stage of the reduction. A QR factorization is computed for the tile $A_{2,2}$ (the red tile). After this QR is finished, a set of independent tasks is released

and they all can be executed in parallel. All tiles $A_{2,\bullet}$ (the black tiles of Figure 3(a)) can be updated by applying the Householder transformations that are generated by the QR factorization of $A_{2,2}$. Also, all the tiles $A_{\bullet,2}$ (the magenta tiles of Figure 3(a)) can also be independently annihilated one after another with the R factor of $A_{2,2}$. After each tile $A_{i,2}$ is annihilated (for example the dark magenta tile of Figure 3(a)), a set of parallel tasks may be launched to update all the tiles of the block row i (the dark yellow tiles of Figure 3(a)). Moreover, when $A_{2,3}$ is updated, then an LQ factorization is performed for this tile (the blue tile of Figure 3(b)). Similarly to the QR process, after LQ, all the tiles in the third column of tiles: $A_{3:nt,3}$ (the grey tiles of Figure 3(b)): can now be independently updated by the Householder vectors from the LQ factorization, provided that they have been updated with the transformation from the QR factorization. Similarly, all the tiles $A_{2,4:nt}$ (the cyan tiles of Figure 3(b)) can also be annihilated. Likewise, each annihilation of $A_{1,i}$ (the dark cyan tile of Figure 3(b)) enables a set of tasks to update the block column i (the dark yellow tiles of Figure 3(b)). We note, that this requires implementations of new computational kernels to be able to operate on the new data structures. The details of the implementation of this stage are provided elsewhere [34, 55]. This process of interleaving the QR and the LQ factorizations at each step repeats until the end, and, as result, we obtain a band matrix with a bandwidth of size nb. As we mentioned above, the tile formulation of the algorithm resulted in creation of a large number of parallel tasks. These tasks are organized into a directed acyclic graph (DAG) [11, 15], with the nodes representing the computational tasks and the edges—the data dependencies between them. Thus, restructuring of the linear algebra algorithms as a sequence of tasks that operate on tiles of data removes the fork-join bottleneck that befits the LAPACK-style implementations. It also avoids idle time for individual cores, while, at the same time, increasing the data locality for each core. Moreover, in order to increase the efficiency of our algorithm, we improved our dynamic scheduler by developing two main features that played an important role in to targetting high-performance execution rates. Two auxiliary options were added to the scheduler that allow to label some of the tasks `PRIORITY` and `LOCALITY`. For example, when the QR factorization of $A_{1,1}$ and a sequence of QR updates ensues, it is better to increase the priority of all the tasks that modify the tile $A_{1,2}$, in such a way that the LQ process of $A_{1,2}$ start as soon as possible. As a result, this technique will increase the number of parallel tasks and will aid the interleaving of the tasks from both the QR and the LQ update sequences. As described below, this has a big impact on data locality. The second task label that we developed is the `LOCALITY` flag. It is used for the update of any tile of $A_{2:n,2:n}$. It makes it possible for $A_{2,2}$ to be updated by the QR's Householder process and by the LQ Householder process. Hence, our scheduler has an opportunity to let the same core update $A_{2,2}$ by the two transformations one after the other. This will result in the tile data being loaded from the main memory only once.

5.2 Second Stage: Cache-Friendly Computational Kernels

The band form is further reduced to the final condensed form using the bulge chasing technique. This procedure annihilates the extra off-diagonal elements by chasing the created fill-in elements down to the bottom right side of the matrix using successive orthogonal transformations at each sweep. This stage involves memory-bound operations and requires the band matrix to be accessed from multiple disjoint locations. In other words, there is an accumulation of substantial latency overhead each time different portions of the matrix are loaded into cache memory, which is not compensated for



(a) xGBCW1 (green). (b) xGBCW2 (red).
(c) xGBCW3 (green). (d) bulge overlap.

Figure 4: Kernel execution of the BRD algorithm during the second stage.

by the low execution rate of the actual computations (the so-called surface-to-volume effect). To overcome these critical limitations, we developed a bulge chasing algorithm, very similar to the novel bulge chasing techniques for symmetric eigenvalue problems (reduction from band to bidiagonal) developed in [34], but we differ from it in using a column-wise elimination instead of an element-wise elimination. In addition, we also differ by developing our kernels to deal with general matrices instead of symmetric matrices. When the singular vectors need to be computed, the most problematic aspect of the standard procedure is the element-wise elimination [34]. Such an implementation is very suitable when only singular value is required, but is limited when singular vectors are required. In particular, it generates element-wise Householder reflectors, and thus, the update of the singular vectors by these reflectors becomes the bottleneck as it is based on BLAS 1 operations. Our modification adds a small amount of extra work but it allows the use of the Level 3 BLAS kernels to compute the transformations or to apply them in the form of the orthogonal matrix U_2 and V_2 – the result of computation in this phase. Moreover, we designed our algorithm to extensively use cache friendly kernels combined with fine grained, memory aware tasks in an out-of-order scheduling technique which considerably enhances data locality. The purpose of this section is only to make the paper self-contained. Therefore, we only briefly describe our column-wise bulge chasing approach for general band matrices as well as the technique used for task scheduling and increasing data locality [35].

The bulge chasing algorithm consists of a succession of three new kernels. These kernels have been designed to increase cache reuse. The idea is to load a block of data in the actual cache memory and to apply all the possible computation to it before unloading it. The first kernel called xGBCW1 illustrated in Figure 4(a), manipulate the green block of data. It triggers the beginning of each sweep

by annihilating the extra non-zero entries within a single row, then applies the computed elementary Householder reflector from the right within this block. Hence, it subsequently generates triangular bulges as shown in Figure 4(a) (the black block). Note that this triangular bulge must be annihilated in order to eventually avoid the excessive growth of the fill-in structure. A classical implementation will eliminate the whole triangular bulge. However, for an appropriate study of the bulge chasing procedure, let us remark that the elimination of the row $i+1$ (the sweep $i+1$), at the next step, creates a triangular bulge which will overlap this one by one column shift to the right and one row to the bottom, as shown in Figure 4(d) where the reader can see that the lower triangular portion of the blue block (the bulge created in sweep $i+1$) overlaps with the lower triangular portion of the green block (corresponding to the bulges created by the previous sweep i). As a result, we can reduce the computational cost and instead of eliminating the whole triangular bulge created for sweep i, we only eliminate the non-overlapped region of it: its first column. The remaining columns can be delayed to the upcoming annihilation sweeps. In this way, we can avoid the growth of the bulges and reduce the extra cost accrued when the whole bulge is eliminated. Moreover, we designed a cache friendly kernel that takes advantage of the fact that the created bulge (the green block) remains in the cache and therefore it directly eliminates its first column and applies the corresponding left update to the remaining column of the green block. The second kernel, xGBCW2, loads the next block and it applies the necessary left updates derived from the previous kernel. Eventually this will generates triangular bulges as shown in Figure 4(b). Hence, this kernel will also annihilate the first row of the created bulge and update its red block from the right. Finally, the third kernel, xGBCW3, loads the next block (the third green block of Figure 4(c)) and it continues applying, from the right, the transformations of the previous kernel 2. Like kernel 1, it also creates a bulge which is removed and the green block is updated correspondingly from the left similar to the process undertaken by kernel 1. Accordingly, the annihilation of each sweep can be described as a single call to kernel 1 followed by repetitive calls to a cycle of the kernels 2 and 3.

The implementation of this stage is done by using either a dynamic or a static runtime environment that we developed. This stage is, in our opinion, one of the main challenges for algorithms as it is difficult to track the data dependencies. The annihilation of the subsequent sweeps will generate computational tasks, which will partially overlap within the data used by the tasks of the previous sweeps (see Figure 4(d)) – the main challenge of dependence tracking. We have used our data translation layer (DTL) and functional dependencies [34, 55] to handle the dependencies and to provide crucial information to the runtime to achieve the correct scheduling. As mentioned above, the amount of data involved by each task is of size $nb \times nb$ but are handled efficiently because our kernels increase cache reuse. We also developed our scheduling technique to minimize the memory traffic and increase the efficiency of our algorithm in such a way that the subsequent tasks that involve the same region of data will be executed by the same thread. As an example, the first task $T_1^{(2)}$ of the annihilation of sweep 2, which operates on the magenta block of Figure 4(d), overlaps the region of the data that has been used by the first task $T_1^{(1)}$ of the sweep 1 (green block). Yet, it is obvious to execute task $T_1^{(2)}$ by the same thread that executed $T_1^{(1)}$. To ensure the maximum reuse, we force the scheduler to distribute the tasks according to their data location. The main goal of our data-based scheduling is to define fine-grain local tasks that increase cache reuse and minimize communication. For small matrix sizes, we rather prefer the use of a subsequent set of threads

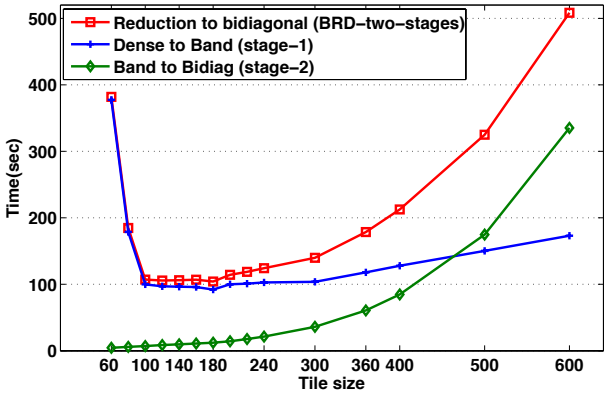


Figure 5: The effect of the tile size on the performance of both stages for a matrix of size 16000 **using** 48 **cores of system A.**

(the number of threads that fit the matrix into its fast memory) while leaving the other threads working on different portions of the code rather than using all the available resources. The implementation of this phase has been well optimized. It has been observed that it takes between 5% to 10% of the global time of the reduction from dense to bidiagonal.

5.3 The Effects of the Bandwidth Size

From the foregoing, it is clear that the tile/band size is the paramount parameter for achieving high performance. As opposed to the first stage kernels, the kernels of the second stage are clearly memory-bound and rely on Level 2 BLAS operations. Their performance depends on how much data can fit in the fast memory. Thus, if the tile size chosen during the first stage is too large, the second stage may encounter significant difficulties coping with the memory bus latency. Figure 5 illustrates the effect of this parameter on the first stage (reduction to general band) blue curve with "+", and on the second stage (the bulge chasing) green curve with "diamond". As expected, on the one hand, the tile size needs to be large ($120 < nb < 300$) to extract high performance from the first stage but not extremely large. Evidently, for $nb > 360$, we lose the gain obtained from the locality and the cache reuse as the data does not fit into the L2 cache level and also we lose a degree of the parallelism as the number of tiles $nt = n/nb$ decreases. On the other hand, the tile size has to be small enough to extract high performance (thanks to the cache speed up) from the bulge chasing stage. Thus a trade-off between them is one of the best choices. In our experiment we found that $120 < nb < 200$ looks to be the best compromise.

5.4 Application of Matrices for the Orthogonal Transformation

The standard one-stage approach reduces the dense matrix A to bidiagonal B and computes its singular values and vectors as mentioned earlier in Section 1. The singular vectors of A: U and V^H; are computed by updating of the singular vectors of B from the left by Q and from the right by P^H, respectively. In the case of the two-stage approach, the first stage reduces the original general dense matrix A to a general band matrix by applying a two-sided transformations to A such that $Q_1^H A P_1 = B$. Similarly, the second stage—the bulge-chasing—reduces the band matrix B to the bidiagonal form by applying the transformation from both the left and the right side to B such that $Q_2^H B P_2 = bi$. Let's now denote the SVD of A by $A = U\Sigma V^H$, where U and V^H are, respectively, the left and the right singular vectors of A, and the diagonal entries of Σ hold the

singular values. These singular vectors must be updated by both of Q_* and P_*, according to (1)

$$\begin{array}{lll} U & = Q_1 Q_2 L_s & = (I - V_1 T_1 V_1^H)(I - V_2 T_2 V_2^H) Ls, \\ and, & & \\ V^H & = R_s P_2^H P_1^H & = R_s (I - W_2 Tr_2^H W_2^H)(I - W_1 Tr_1^H W_1^H) \end{array} \quad (1)$$

where (V_1, T_1 and W_1, Tr_1) and (V_2, T_2 and W_2, Tr_2) represent the left and the right Householder reflectors generated during the first and the second stages of the reduction to the bidiagonal form. L and R are the left and the right singular vectors of the bidiagonal matrix $bi = L_s \Sigma R_s$. It is clear that the two-stage approach introduces non-trivial amount of extra computation: the application of Q_2 and P_2^H; for the case when the singular vectors are needed. Thus, one of the crucial procedures of the two-stage algorithm, when optimizing for performance, is the update of the singular vectors by the Householder transformations. The transformations that were generated during the two stages of the reduction to the bidiagonal form. Obviously, the implementation of this scheme is not as straightforward as simply parallelizing a loop. In particular, because of complications of the bulge-chasing mechanism, the order of generated task dependencies is quite intricate.

5.5 The Effects of Blocking

The application of the Householder reflectors V_2, W_2 is very challenging and is usually the most time consuming operation when applied in the standard fashion. We present the structure of V_2 in Figure 6(a)—the structure of W_2 is similar. Note that these reflectors represent the annihilation of the band matrix, and thus each is of length nb – the bandwidth size. Let's assume that we want to apply V_2 to the left singular vectors L_s: $(I - V_2 T_2 V_2^H) L_s$. The standard procedure does not offer parallelism and is memory-bound because it relies on updating L_s from the left by each of the v_2 in sequence. The parallelism here is expressed by the number of v_2 within a column, which is n/nb. All the operations are based on Level 2 BLAS and require the corresponding row of the matrix L_s to be loaded into memory for each reflector v_2. One may conclude that such an approach produces poor results. In Figure 7, we depict the execution time in seconds to performs the $Q_2 L_s$ update using this approach. As expected, the performance shown in Figure 7 is very poor and is shown only as a baseline. A better procedure is to update with calls to Level 3 BLAS, which achieves both: very good scalability and performance. The priority is to create compute intensive operations to take advantage of the efficiency of Level 3 BLAS. We proposed and implemented accumulation and combination of the Householder reflectors. This is not always easy, and to achieve this goal we must pay attention to the overlap between them as well as the fact that their application must follow the specific dependency order of the bulge chasing procedure in which they have been created. To stress on these issues, let's clarify it more by giving this example. For sweep i (e.g., the column at position B(i,i):B(i+nb,i)), its annihilation generates a set of k Householder reflectors (v_i^k, w_i^k), each of length nb, the v_i^k are represented in column i of the matrix V_2 depicted in Figure 6(a). Likewise, the ones related to the annihilation of sweep $i+1$, are those presented in column $i+1$, where they are shifted one element down compared to those of sweep i. We believe there is a way to implement a blocked version for applying the V's while adhering the constraints defined above. It is accomplished by combining of the reflectors v_i^k from sweep i with those from sweep $i+1$, $i+2$,..., $i+u$ that follow the diamond shape region as defined in Figure 6(a). While each of those diamonds is considered as one block, their application needs to follow the dependency order. For example, applying the green block 4 and the red block 5 of

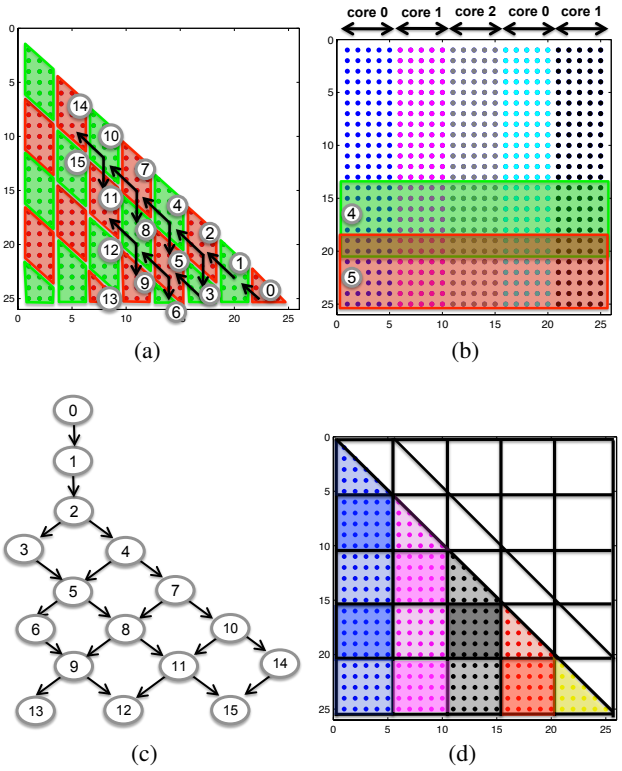


Figure 6: (a) Tiling of V_1, (b) Blocking technique to apply V_2, (c) Distribution of the singular vectors matrix that creates independent fashion of applying V_2 which increase locality per core, (d) Portion of the DAG showing the dependency of the V's of V_2.

the V_2's in Figure 6(a) modifies the green block row 4 and the red block row 5, respectively, of the singular vector matrix L_s drawn in Figure 6(b) where one can easily observe the overlapped region. Referring to the chasing order, block 4 needs to be applied before block 5. We generate a portion of the dependency graph (DAG) that expresses the connections between the successive tasks involved in the application of the V's, and illustrate it in Figure 6(c).

For simplicity, we also draw a sample of those dependencies by the arrows in Figure 6(a). The parallel computation that can be obtained corresponds to the separated leaf node of the tree of Figure 6(c). This process leads to a very limited number of parallel and pipelined tasks. However, if we take advantage of the left application of the Householder reflectors $Q_2 L_s$ being completely independent in respect to the column of L_s, then we can express a large number of independent parallel tasks by splitting L_s by a block of columns over the number of threads as shown in Figure 6(b), where each diamond block can be applied independently to each portion of L_s. This two-way of parallelism can be considered adequate for the development of high performance algorithms. Moreover, this method does not require any data communication between cores. The overlap between each application of V's as described above increases the cache reuse. We also define the size of each block of L_s in a way to fit more than one region of it in the L2 cache to increase data locality. We implemented a new kernel that deals with these diamond shapes in a way that increases the cache reuse. We report the improvements obtained by the blocking technique that is combined with the two-way parallelism in Figure 7. The results

show the attractive improvement achieved here. More details will be discussed in the experimental section.

The application of the first stage reflectors V_1 to the resulting matrix $G = (I - V_2 T_2 V_2^H) L_s$ proceeds as follows. The V_1's reflectors are stored in a tile storage fashion to increase the data locality. and we illustrate its structure in Figure 6(d). The parallelism of this step can be expressed in the two following descriptions. On the one hand, thanks to the tile algorithm, this procedure inherits parallelism and it is well adapted for parallel computing. Here each tile represents a block of V_1's where its application invokes different areas of the matrix G, and so can be applied independently, e.g., any tile of the magenta column of Figure 6(d) modifies different areas of G and can proceed in parallel. The only constraint to satisfy is to keep the left order update, meaning that as soon as the $v_{4,3}$ (black tile(4,3)) updates the matrix G, the latter could be updated by the $v_{4,2}$ and then by the blue $v_{4,1}$.

On the other hand, similarly the singular vectors matrix G can also by partitioned into a set of independent blocks/tiles to be updated. As a result, the design of the tile algorithm using the two-way parallelism generates a large number of independent tasks that can be applied in an asynchronous manner using either a static or dynamic scheduler. Finally, since all v tiles of a block-row of V_1 update the same blocks of G, then the data-locality of all these tasks is forced to be executed on the same thread.

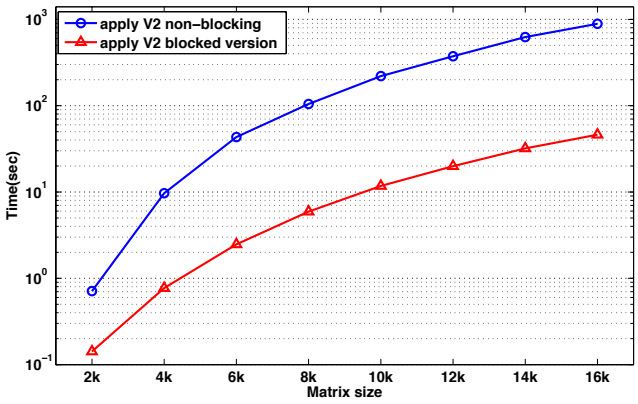


Figure 7: The effect of a blocked v.s., non-blocked version of the application of Householder reflectors V_2 using 48 cores of system A.

5.6 Trading Extra Computation for Higher Performance Rate

The implementation that is discussed here is more related to the hardware architecture based on hierarchical memory. The blocking technique described for the application of the Householder reflectors generated during the bulge chasing stage (V_2, W_2) requires more sophistication since it might increase the performance of this procedure. As described above, the size of each block of the reflectors $V_2^{(k)}$ is $(nb+u) \times u$ where u is the blocking factor used. Because of the diamond shape of the $V_2^{(k)}$ blocks, each multiplication by a block of reflectors $V_2^{(k)}$ proceeds as three operations: 1) a *TRMM* multiplication by the top lower triangular portion of $V_2^{(k)}$ of size $u \times u$; 2) a *GEMM* with the middle portion of size $(nb-2u) \times u$, and 3) another *TRMM* by the bottom upper triangular portion of $V_2^{(k)}$ of size $u \times u$. We believe that, for computational efficiency, we might replace all the *TRMM* by *GEMM* and thus, it performs only a one *GEMM* with $V_2^{(k)}$ of size $(nb+u) \times u$, especially that *GEMM*'s operation are more advantageous when dealing with small sizes. To illustrate this, we perform experiments with both version and plot results on Figure 8. The improvement that is obtained is more than 10% especially for large size where a large number of multiplication by these small diamond blocks is performed.

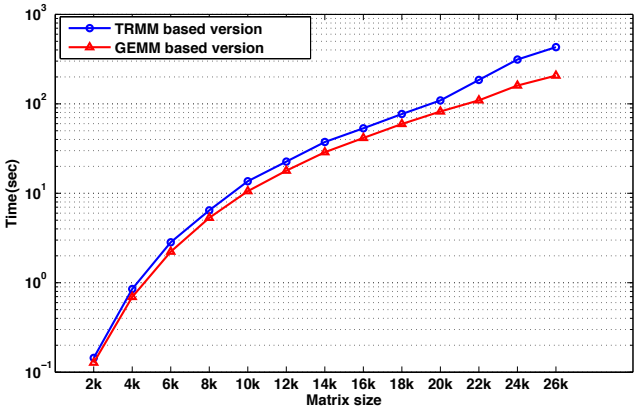


Figure 8: The effect of trading one Level 3 BLAS operation for another Level 3 BLAS with different computational complexity.

6. EXPERIMENTAL RESULTS

This Section presents the performance comparisons of our tile algorithm for two-stage SVD against the state-of-the-art numerical linear algebra libraries.

6.1 Experimental Environment

Our experiments have been performed on two shared memory systems. They are representative servers-grade machines as well as workstations commonly used for computationally intensive workloads. The first system named system A is composed of a four-socket AMD Opteron(tm) 6180 SE: 12 cores each (48 cores total), running at 2.5 GHz with 128 GiB of main memory; the total number of cores is evenly spread among two physical mother boards. The level 2 cache size per core is 512 KiB. These computations are done in double precision arithmetic. The theoretical peak for this architecture in double precision is 480 Gflop/s (10.1 Gflop/s per core). Our second system named system B is an Intel multicore system with dual-socket, 8 core Intel Xeon E5-2670 (Sandy Bridge) processors, each running at 2.6 GHz. Each socket has a 24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1. The system is equipped with 52 Gbytes of memory. The theoretical peak for this architecture in double precision is 20.8 Gflop/s per core. There are a number of software packages that include a singular value decomposition solver. For comparison, we used the latest MKL (Math Kernel Library) [43] version 13.1, which is a commercial library from Intel that is highly optimized for Intel processors and performs competes favorably with alternatives on AMD processors. It includes a comprehensive set of mathematical routines implemented to run well on most x86 multicore processors. In particular, MKL includes LAPACK-equivalent[1] routines to compute the bidiagonal reduction

[1]We consider a routine to be LAPACK-equivalent if it provides the same behavior in terms of numerical error bounds and can handle the same input parameters.

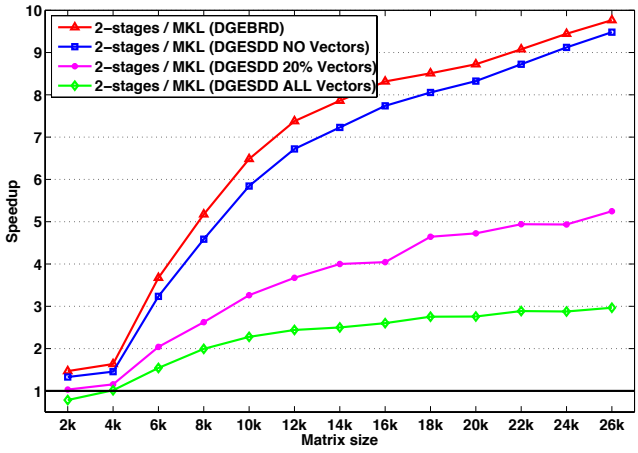


Figure 9: The speedup obtained by our implementation of DGESDD versus its counterpart from the Intel MKL library on 48 AMD cores of system A.

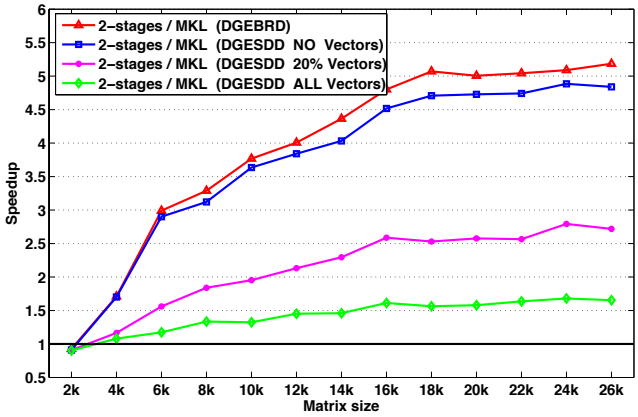


Figure 10: The speedup obtained by our implementation of DGESDD versus its counterpart from the Intel MKL library on 16 Sandy Bridge cores of system B.

DGEBRD, and routines to find the singular value decomposition such as DGESDD (the divide and conquer **D&C** algorithm) and DGESVD (the implicit zero-shift QR algorithm).

6.2 Performance results

The following experiments illustrate the superior efficiency and the scalability of our proposed SVD solver with respect to the state-of-the-art optimized vendors numerical linear algebra libraries. Each of every graph below presents many comparison curves that we will describe. We performed an extensive study with a large number of experimental tests on two different machines (one with large number of cores system A and another with small number of cores system B) in order to give the reader as much information as possible. We computed the SVD decomposition, where we either compute only the singular values, or both singular values and vectors, with two different algorithms (DGESVD, DGESDD), varying the size of matrices from 2000 to 26000 using our two system (the 48 AMD cores of system A and the 16 Sandy Bridge cores of system B). As many applications need only a portion of the singular vectors, we also present results using the DGESDD (the divide and conquer SVD solver) to compute a 20% of the singular vectors. In addition, in order to make our experiments self-contained, we report the result of the improvements that our two-stage implementation brought to the reduction to bidiagonal form compared against the one-stage approach from the state-of-the-art numerical linear algebra libraries.

In particular, Figure 9 and Figure 10, displays speedups and efficiencies for computing the SVD using the divide and conquer technique. For each speedup curve (representing a routine), we depict the ratio of the running time between the routine from the Intel's MKL library and its counterpart from our implementation within the same computing environment. These results show four types of behavior.

Let's first comment on the reduction to bidiagonal form (DGEBRD: the red curve) and on the SVD decomposition when only the singular values are computed (DGESDD NO Vectors: the blue curve). The speedups shown are remarkable, our implementation asymptotically achieves more than 8× speedup on the 48 cores of system A and more than 4× speedup on the 16 cores of system B. We had expected such improvement. The results obtained here confirm the importance of the considerations discussed in our study, when designing high performance libraries. The gain is due to the tile algorithm that we developed and to its efficient implementation of the first stage (reduction to band) which is the compute intensive stage, and from the design of both the second stage (bulge chasing) and the orthogonal transformation update, that maps both the algorithm and the data to the hardware using cache friendly kernels and scheduling based on increasing data locality. Our new algorithm scales as the matrix sizes increase and asymptoticly achieves a perfect speedup, despite the side effects of running on a NUMA system.

In contrast to the standard approach where the reduction to bidiagonal dominates the overall time (as described above in Figure 2), the two-stages reduction to bidiagonal consists of less than 20% of the overall time. Thus, when only singular values or small portion of the vectors are needed, our approach is particularly favorable. One of the most fruitful advantage of our two-stages SVD algorithm is the attractive speedup shown when a portion of the singular vectors is computed. The results obtained by the DGESDD when 20% of the singular vectors are required are more than 4× faster when using the 48 cores of system A (magenta curve of Figure 9) and can get more than 2× speedup when using the 16 cores of system B (magenta curve of Figure 10). The underlying results are straightforward. Computing portion of the singular vectors significantly minimizes the overhead of the extra cost (applying Q_2 and P_2) introduced by the two-stages approach, and therefore the speedup observed here is relatively high.

Moreover, we also evaluated the performance and the speedup of our algorithm when all the singular vectors are computed (green curve with diamond of Figure 9 and 10). We can observe that our algorithm performs consistently better than the state-of-the-art optimized MKL routine DGESDD with the exception of small matrices. Our implementation is around three times faster when using large number of cores (48 cores of system A) and get around 1.5 speedup on small number of cores (16 cores of system B).

Our fourth observation is related to the scalability of our implementation. To explain it briefly, we have performed experiments with 12 cores on system A, and compare it with the presented results on the 48 cores. Our solver accomplished very good scalability, the execution time on 48 cores is about 3.5 times faster than the one obtained on 12 cores. It also appears from the speedup illustrated in our figures that the scalability increases with respect to the number

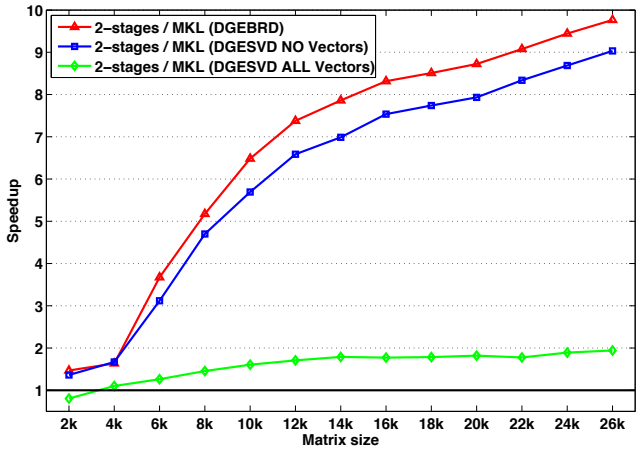


Figure 11: The speedup obtained by our implementation of DGESVD versus its counterpart from the Intel MKL library on 48 AMD cores of system A.

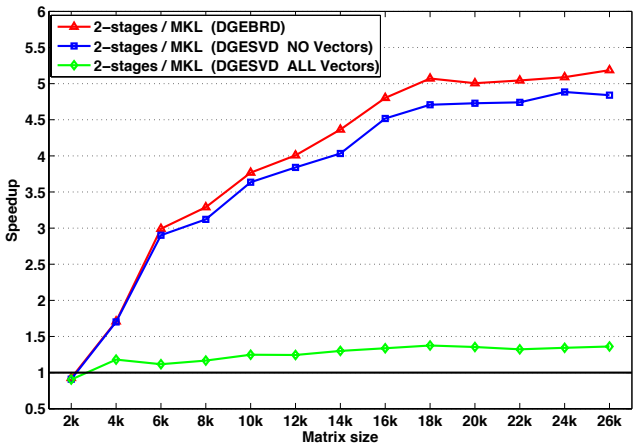


Figure 12: The speedup obtained by our implementation of DGESVD versus its counterpart from the Intel MKL library on 16 Sandy Bridge cores of system B.

of cores, and for that we believe that this implementation is suitable to exploit distributed and large shared machines.

Since we prefer to cover all the available SVD-solver and not be limited by the divide and conquer, we also performed the same set of experiments using the implicit zero-shift QR algorithm (DGESVD) as another solver variant. The speedup results obtained on both of our machines system A and system B, are presented in Figure 11 and Figure 12, respectively. We note that, here we couldn't compute a set of the singular vectors because the routine which computes the SVD decomposition of the bidiagonal matrix (DBDSQR) does not provide this option and requires that the transformation matrices Q and P^H be explicitly provided as input. However, experiments with either all the singular vectors or none of the singular vectors were reported. The speedup of the DGEBRD routine has been reported in Figure 11 and Figure 12, for the completeness of our graphs.

The performance shown in Figure 11 and Figure 12, is very close to the ones using the divide and conquer solver. It drops slightly compared to Figure 9 and Figure 10, because the DBDSQR solver used here requires the matrix Q and P explicitly computed in input, and so it update the singular vectors L_s and R_s internally. Thus it does not benefit from all the optimizations we implemented. However, the explicit generation of the matrices Q and P take advantage of all the techniques described in this paper. As a consequence, the trend of the speedup curve of Figure 11 and Figure 12 is slightly decreases compared to the one of Figure 9 and Figure 10. Therefore, reaching a two-fold speedup (Figure 11) is worthy effort and can be considered speed-up by more than a factor of 2.

Finally, we demonstrate that our algorithm is very efficient and can achieve more than two-fold speedup over the well know state-of-the-art optimized libraries. It is also well suitable especially when only the singular values or when a portion of the singular vectors is needed – the results show $4\times$ to $10\times$ speedup. We believe that this achievement makes our algorithm a very good candidate for the current and next generation of machines.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel implementation of an algorithm that computes singular values and vectors of a dense matrix. Our algorithm is based on the two-stage approach and thus performs twice as many floating operations to obtain the singular vectors when compared with the classic approach that is currently in common use. Such drastic increase in operation count might have been considered a hindrance a few years back, but on modern hardware it is not the case. We attribute this to the formulation of the algorithm in terms much more efficient kernel routines, and we show how the benefit both theoretically as well as through practical experiments. Instead of a two-fold slow-down that would have been expected from the two-fold increase in the operation count, we were able to achieve more than two-fold speed-up over the current breed of state-of-the-art software packages that were considered the fastest at the time of this writing. The performance obtained is very encouraging. We believe that these techniques will only increase in relevance for up-coming architectures. Additionally, because of good scalability properties of our algorithm, we believe that our approach lends itself well to distributed memory implementations and we plan to pursue this direction in the future.

8. ACKNOWLEDGMENTS

The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA, and MathWorks for supporting this research effort.

9. REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009. DOI: 10.1088/1742-6596/180/1/012037.

[2] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[3] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. `http://www.netlib.org/lapack/lug/`.

[4] H. C. Andrews and C. L. Patterson. Singular value decompositions and digital image processing. *IEEE Trans Acoust., Speech, Signal Processing ASSP-24*, 1, February 1976.

[5] R. H. Bartels, G. H. Golub, and M. Saunders. Numerical techniques in mathematical programming. In *Nonhnear Programming*, pages 123–176, New York, 1971. Academm Press.

[6] M. Bečka, G. Okša, and M. Vajteršic. Parallel Block-Jacobi SVD. In M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied, editors, *Methods High-Performance Scientific Computing*, pages 185–197. Springer, London Dordrecht Heidelberg New York, 2012. ISBN 978-1-4471-2436-8 e-ISBN 978-1-4471-2437-5 DOI 10.1007/978-1-4471-2437-5.

[7] C. Bischof, B. Lang, and X. Sun. Parallel tridiagonalization through two-step band reduction. In *In Proceedings of the Scalable High-Performance Computing Conference*, pages 23–27. IEEE Computer Society Press, 1994.

[8] C. H. Bischof, B. Lang, and X. Sun. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM TOMS*, 26(4):602–616, 2000.

[9] C. H. Bischof and C. V. Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Statist. Comput.*, 8:s2–s13, 1987.

[10] N. Bregman, R. Bailey, and C. Chapman. Ghosts in tomography: the effects of poor angular coverage in 2-D seismic traveltime inversion. *Can. J. Explor. Geophys*, 25(1):7–27, 1989.

[11] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2006.

[12] A. Buttari, J. J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick. Multithreading for synchronization tolerance in matrix factorization. In *Scientific Discovery through Advanced Computing, SciDAC 2007*, Boston, MA, June 24-28 2007. Journal of Physics: Conference Series 78:012028, IOP Publishing. DOI: 10.1088/1742-6596/78/1/012028.

[13] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008. DOI: 10.1002/cpe.1301.

[14] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parellel Comput. Syst. Appl.*, 35:38–53, 2009. DOI: 10.1016/j.parco.2008.10.002.

[15] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 116–125, New York, NY, USA, 2007. ACM.

[16] T. A. Davis and S. Rajamanickam. Algorithm 8xx: PIRO BAND, Pipelined Plane Rotations for Blocked Band Reduction. *Submitted to ACM TOMS, Available at* `www.cise.ufl.edu/~srajaman/genband.pdf`, 2010.

[17] P. Deift, J. W. Demmel, L.-C. Li, and C. Tomei. The bidiagonal singular value decomposition and Hamiltonian mechanics. *SIAM J. Numer. Anal.*, 28(5):1463–1516, October 1991. (LAPACK Working Note #11).

[18] J. W. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, September 1990. (Also LAPACK LAWN #3).

[19] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. In *ParCo 2011 – International Conference on Parallel Computing*, Ghent, Belgium, August 30-September 2 2011.

[20] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. High performance matrix inversion based on LU factorization for multicore architectures. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, MTAGS '11, pages 33–42, New York, NY, USA, 2011. ACM.

[21] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. *Advances in Parallel Computing, Special Issue*, 22:429–436, 2012. ISBN 978-1-61499-040-6 (print); ISBN 978-1-61499-041-3 (online).

[22] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215 – 227, 1989.

[23] V. Farra and R. Madariaga. Seismic waveform modeling in heterogeneous media by ray perturbation theory. *Journal of Geophysical Research: Solid Earth*, 92(B3):2697–2712, 1987.

[24] V. Fernando and B. Parlett. Accurate singular values and differential qd algorithms. *Numerisch Math.*, 67:191–229, 1994.

[25] W. Gansterer, D. Kvasnicka, and C. Ueberhuber. Multi-sweep algorithms for the symmetric eigenproblem. In *Vector and Parallel Processing - VECPAR'98*, volume 1573 of *Lecture Notes in Computer Science*, pages 20–28. Springer, 1999.

[26] G. H. Golub and W. Kahan. Calculating the singular values and pseudoinverse of a matrix. *SIAM J. Numer. Anal.*, 2(3):205–224, 1965.

[27] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The John Hopkins University Press, 4th edition, December 27 2012. ISBN-10: 1421407949, ISBN-13: 978-1421407944.

[28] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. In J. Wilkinson and C. Reinsch, editors, *Handbook for Automattc Computation, II, Linear Algebra*. Springer-Verlag, New York, 1971.

[29] G. H. Golub and J. H. Wilkinson. Ill-conditioned eigensystems and the computation of the Jordan canonical form. *SIAM Rev.*, 18(4), October 1976.

[30] R. Grimes, H. Krakauer, J. Lewis, H. Simon, and S.-H. Wei. The solution of large dense generalized eigenvalue problems on the cray X-MP/24 with SSD. *J. Comput. Phys.*, 69:471–481, April 1987.

[31] R. G. Grimes and H. D. Simon. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Transactions on Mathematical Software*, 14:241–256, September 1988.

[32] M. Gu and S. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Mat. Anal. Appl.*, 16:79–92, 1995.

[33] F. G. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion.

ACM Trans. Math. Soft., 38(3):article 17, 2012. DOI: 10.1145/2168773.2168775.

[34] A. Haidar, H. Ltaief, and J. Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of SC '11*, pages 8:1–8:11, New York, NY, USA, 2011. ACM.

[35] A. Haidar, H. Ltaief, P. Luszczek, and J. Dongarra. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Shanghai, China, May 21-25 2012. ISBN 978-1-4673-0975-2.

[36] A. Haidar, H. Ltaief, A. YarKhan, and J. J. Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurrency Computat.: Pract. Exper.*, 2011. DOI: 10.1002/cpe.1829.

[37] A. Haidar, P. Luszczek, and J. Dongarra. New multi-stage algorithm for symmetric eigenvalues and eigenvectors achieves two-fold speedup. In *Euro-Par 2013*, Aachen, Germany, August 26-30 2013. (submitted).

[38] A. Haidar, S. Tomov, J. Dongarra, R. Solca, and T. Schulthess. A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. *International Journal of High Performance Computing Applications*, September 2012. (accepted).

[39] R. J. Hanson. A numerical method for solving Fredholm integral equations of the first kind using singular values. *SIAM J. Numer. Anal.*, 8(3):616–626, 1971.

[40] H. Hotelling. Analysis of a complex of statistical variables into principal components. *J. Educ. Psych.*, 24:417–441, 498–520, 1933.

[41] H. Hotelling. Simplified calculation of principal components. *Psychometrica*, 1:27–35, 1935.

[42] A. S. Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4), October 1958. DOI 10.1145/320941.320947.

[43] Intel. Math Kernel Library.

[44] E. R. Jessup and D. Sorensen. A Parallel Algorithm for Computing the Singular Value Decomposition of a Matrix. *SIAM J. Matrix Anal. Appl.*, 15:530–548, 1994.

[45] E. P. Jiang and M. W. Berry. Information filtering using the Riemannian SVD (R-SVD). In A. Ferreira, J. D. P. Rolim, H. D. Simon, and S.-H. Teng, editors, *Solving Irregularly Structured Problems in Parallel, 5th International Symposium, IRREGULAR 98, Berkeley, California, USA, August 9-11, 1998, Proceedings*, volume 1457 of *Lecture Notes in Computer Science*, pages 386–395. Springer, 1998.

[46] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equation on the CELL processor using Cholesky factorization. *Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008. DOI: TPDS.2007.70813.

[47] J. Kurzak and J. J. Dongarra. QR factorization for the CELL processor. *Scientific Programming*, 00:1–12, 2008. DOI: 10.3233/SPR-2008-0268.

[48] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency Computat.: Pract. Exper.*, 21(1):15–44, 2009. DOI: 10.1002/cpe.1467.

[49] B. Lang. A parallel algorithm for reducing symmetric banded matrices to tridiagonal form. *SIAM J. Sci. Comput.*, 14:1320–1338, November 1993.

[50] B. Lang. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Computing*, 25(7):845–860, 1999.

[51] C. Lawson and R. Hanson. *Solving Least Squares Problems*. Prentice-Hall, Englewood Cliffs, N.J., 1974.

[52] H. Ltaief, J. Kurzak, and J. Dongarra. Parallel band two-sided matrix bidiagonalization for multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 21(4), April 2010.

[53] H. Ltaief, P. Luszczek, and J. Dongarra. High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures. *ACM TOMS*, 39(3), 2013. In publication.

[54] H. Ltaief, P. Luszczek, A. Haidar, and J. Dongarra. Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Proceedings of 9th International Conference, PPAM 2011*, volume 7203, pages 661–670, Torun, Poland, 2012.

[55] P. Luszczek, H. Ltaief, and J. Dongarra. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In *IPDPS 2011: IEEE International Parallel and Distributed Processing Symposium*, Anchorage, Alaska, USA, May 16-20 2011.

[56] B. P. M. Berry, D. Mezher and A. Sameh. Parallel methods for the singular value decomposition. In E. Kontoghiorghes, editor, *Parallel Computing and Statistics*, pages 117–164. Chapman & Hall/CRC, 2006.

[57] B. C. Moore. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE Transactions on Automatic Control*, AC-26(1), February 1981.

[58] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, New York, 1973.

[59] G. W. Stewart. The decompositional approach to matrix computation. *Computing in Science & Engineering*, 2(1):50–59, Jan/Feb 2000. ISSN: 1521-9615; DOI 10.1109/5992.814658.

[60] University of Tennessee Knoxville. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3*, November 2010.

[61] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), Aug. 1990. DOI 10.1145/79173.79181.

[62] P. R. Willems, B. Lang, and C. Vömel. Computing the bidiagonal SVD using multiple relatively robust representations. Technical Report CSD-05-1376, University of California Berkeley, Computer Science Division, 2005. Also available as the LAPACK Working Note 166.

[63] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, April 2011.